A Java Virtual Machine for RISC-V
Porting the Jikes Research VM

Martin Maas, UC Berkeley
(maas@eecs.berkeley.edu)

5th RISC-V Workshop, Nov 30, 2016
Mountain View, CA
Why Do We Need a JVM Port?

Run Java Applications
- Apache ZooKeeper
- Apache Spark
- Hadoop

High-performance production JVM

Academic Research

Easy-to-modify research JVM

Which JVM should we port?
Both!
WHY DO WE NEED A JVM PORT?

Run Java Applications

- High-performance production JVM

OpenJDK/Hotspot JVM

Implementer: Michael Knyszek
Status: Runs with zero backend, C1 port in progress

Academic Research

Easy-to-modify research JVM

Jikes Research VM

Implementer: Martin Maas
Status: Runs with baseline (non-optimizing) JIT compiler
**Why Do We Need a JVM Port?**

**Run Java Applications**
- Apache ZooKeeper™
- Hadoop
- Apache Spark™

**High-performance production JVM**

**OpenJDK/Hotspot JVM**

**Implementer:** Michael Knyszek  
**Status:** Runs with zero backend, C1 port in progress

**Jikes Research VM**

**Implementer:** Martin Maas  
**Status:** Runs with baseline (non-optimizing) JIT compiler

**Academic Research**

Easy-to-modify research JVM
1. The Jikes Research VM
   How does it work, what does it do?

2. Porting a JVM to RISC-V
   Lessons learned from porting a JVM

3. Status & Next Steps
   Screenshots and current status of the port
PART I
The Jikes Research VM
Originally written at IBM in 1997, maintained by research groups (ANU, UT Austin, etc.)

Used in **over 200 research papers**
- 2012: ACM SIGPLAN Software Award

**JikesRVM is a JVM written in Java** (this is also called a “metacircular” runtime system)
A JVM written in Java, how does this work?

Special classes encapsulate low-level primitives:
• Regular classes for the bootstrap JVM
• Handled specially by JikesRVM JIT compiler
A JVM written in Java, how does this work?

**Step 1:** Load JikesRVM into itself

**Step 2:** JIT compiler produces code and stores it to memory using machine-level primitives

**Step 3:** Store Image to disk
Running the JVM (without the bootstrap JVM)

Image

Bootloader (C code)

JikesRVM

Java Program

In JIT-generated code, primitives map to actual operations

Resume at “boot” function.

Loads boot image into memory implements native calls, fault handlers, etc.
JikesRVM allows writing systems code in a type-safe high-level language, and is easy to modify. This makes it perfect for research!
PART II
Porting a JVM to RISC-V
JVMs have a large number of dependencies. Use the `riscv-poky` Linux distribution generator to build a cross-compile SDK and a Linux image.

Clone `riscv-poky` and run:

```
bitbake meta-toolchain
```

To add missing dependencies, edit:

```
meta-riscv/recipes-core/images/core-image-riscv.bb
```

Install the resulting SDK image.
LESSON #2: BUILD PROCESS

For fast turnaround, write a single script that:

(1) Builds the JVM and copies the result into the Linux image
(2) Boots the JVM in QEMU with an inittab that runs the JVM, copies the result into a file and shuts down the machine

```
JIKES_PATH=/[…]/BaseBaseNoGC_riscv64-linux
sudo mount -t ext2 -o loop,rw jikes.rootfs.ext2 mount
pushd mount
sudo rm -r home/root/BaseBaseMarkSweep_riscv64-linux
rm $JIKES_PATH/BootImageWriterOutput.txt
sudo cp -R $JIKES_PATH home/root/BaseBaseMarkSweep_riscv64-linux
popd
sudo unmount mount

/etc/inittab
[…]
l6:6:wait:/etc/init.d/rc 6
z6:6:respawn:/sbin/sulogin
H0:12345:respawn:/bin/bash -c "/home/root/test.sh"

/home/root/test.sh
export LD_LIBRARY_PATH=/home/root
echo "====================== […] ======================="
echo "-- JIKES RVM TEST ENVIRONMENT --"
export LD_LIBRARY_PATH=/home/root
cd /home/root/BaseBaseMarkSweep_riscv64-linux/
./rvm -X:verbose -X:verboseBoot=10 HelloWorld 2>&1 | ../spike-dasm
echo "====================== […] ======================="
halt
bash
```
LESSON #3: GENERATE THE ASSEMBLER

Don’t write the assembler by hand, use riscv-opcodes and generate it!
Add assertions to check that constants fit into the immediate, loaded values are correct, etc.

```java
public static void patchCode(ArchitectureSpecific.CodeArray code, int indexa, int indexb) {
    if (VM.VerifyAssertions) VM._assert(VM.NOT_REACHED);
}

public final void check_fits_signed(int val, int bits) {
    if (VM.VerifyAssertions) {
        val = val >> bits - 1;
        VM._assert(val == 0 || val == -1);
    }
}

public final void check_fits_unsigned(int bits, int val) {
    if (VM.VerifyAssertions) {
        VM._assert(val <= 0 && (val >> bits) == 0);
    }
}

public final int mask(int val, int lo, int hi) { return ((val >> lo) & ((1 << (hi-lo+1)) - 1)); }

public final int make_imm20(int val) { check_fits_unsigned(20, val); return (val << 12); }

public final int make_imm12(int val) { check_fits_signed(12, val); return (val << 20); }

public final int make_jimm20(int val) { check_fits_signed(20, val); return (mask(val, 20, 20) << 31) | (mask(val, 1, 10) << 21) | (mask(val, 12, 19) << 12) | (mask(val, 11, 11) << 20); }

public final int make_jimm12hi(int val) { check_fits_signed(12, val); return (mask(val, 12, 12) << 31) | (mask(val, 5, 10) << 25) | (mask(val, 1, 4) << 8) | (mask(val, 11, 11) << 7); }

public final int make_jimm12hi(int val) { check_fits_signed(12, val); return (mask(val, 5, 11) << 25) | (mask(val, 0, 4) << 7); }

public final int make_shamt(int val) { check_fits_signed(5, val); return (val << 20); }

public final int make_shamtw(int val) { check_fits_signed(20, val); return (val << 20); }
```

Before assertions can be caught, let them emit an instruction that causes a SEGFAULT and dump state in the fault handler e.g., LW X0, 4(X0)
Lesson #5: Debug Traces

Generate traces of bytecodes and machine code. Can do this with software breakpoints.

At the start of every Bytecode, emit a text sequence:
LD X0, 1024(X0) # SEGFAULT
(Number of instructions)
(Opcode)
(Stack Offset)

Print out instructions as DASM(0x12345678) and pipe the output through spike-dasm
Lesson #6: Implement JIT Incrementally

- Switch off optional features: No GC, no biased locking, one thread, avoid floating point, etc.
- Put a failing assertion for everything you skip.

Find out what the first bytecode ever executed by the JVM is. Write all your initial tests there.

(For Hotspot, it’s the static initializer of the java.lang.String class, for Jikes it’s VM.boot())
Lesson #7: Calling Convention

RVG is a good starting point (need to intermix C and JVM frames for native calls). Be careful with registers used by glibc (e.g., TP/x4, GP/x3).

```c
GPR[] LOCAL_GPRS = { GPR.X9, GPR.X18, GPR.X19, GPR.X20, GPR.X21, GPR.X22, GPR.X23, GPR.X24, GPR.X25, GPR.X26, GPR.X27 };  
int NUM_LOCAL_GPRS = LOCAL_GPRS.length;

FPR[] LOCAL_FPRS = { FPR.F8, FPR.F9, FPR.F18, FPR.F19, FPR.F20, FPR.F21, FPR.F22, FPR.F23, FPR.F24, FPR.F25, FPR.F26, FPR.F27 };  
int NUM_LOCAL_FPRS = LOCAL_FPRS.length;

// Treat SP and FP special (even though nonvolatile in theory)
GPR[] NONVOLATILE_GPRS = { GPR.X9, GPR.X18, GPR.X19, GPR.X20, GPR.X21, GPR.X22, GPR.X23, GPR.X24, GPR.X25, GPR.X26, GPR.X27 };  
FPR[] NONVOLATILE_FPRS = { FPR.F8, FPR.F9, FPR.F18, FPR.F19, FPR.F20, FPR.F21, FPR.F22, FPR.F23, FPR.F24, FPR.F25, FPR.F26, FPR.F27 };  
int NUM_NONVOLATILE_GPRS = NONVOLATILE_GPRS.length;
int NUM_NONVOLATILE_FPRS = NONVOLATILE_FPRS.length;

GPR[] PARAMETER_GPRS = { GPR.X10, GPR.X11, GPR.X12, GPR.X13, GPR.X14, GPR.X15, GPR.X16, GPR.X17 };  
int NUM_PARAMETER_GPRS = PARAMETER_GPRS.length;

FPR[] PARAMETER_FPRS = { FPR.F10, FPR.F11, FPR.F12, FPR.F13, FPR.F14, FPR.F15, FPR.F16, FPR.F17 };  
int NUM_PARAMETER_FPRS = PARAMETER_FPRS.length;

GPR[] OS_PARAMETER_GPRS = { GPR.X10, GPR.X11, GPR.X12, GPR.X13, GPR.X14, GPR.X15, GPR.X16, GPR.X17 };  
int NUM_OS_PARAMETER_GPRS = OS_PARAMETER_GPRS.length;

FPR[] OS_PARAMETER_FPRS = { FPR.F10, FPR.F11, FPR.F12, FPR.F13, FPR.F14, FPR.F15, FPR.F16, FPR.F17 };  
int NUM_OS_PARAMETER_FPRS = OS_PARAMETER_FPRS.length;
```
Get exceptions to work as quickly as possible!

Exceptions drop us back into bootloader. Implement exception delivery mechanism early for stack traces!

Trigger exceptions deliberately as part of runtime-checks (e.g., Divide-by-zero, range checks)

```java
public final void emitTrapArrayIndex() {
    emitLB(ZERO, ZERO, TrapConstants.TRAP_ARRAY_INDEX);
}

public final void emitTrapCheckCast() {
    emitLB(ZERO, ZERO, TrapConstants.TRAP_CHECKCAST);
}

public final void emitTrapDivideByZero() {
    emitLB(ZERO, ZERO, TrapConstants.TRAP_DIVIDE_BY_ZERO);
}
```
RUNNING “HELLO WORLD”

Booting JikesRVM requires:

- Loading and initializing 93 classes
- Support for most Java bytecodes
- Dynamic linking
- Class loading
- JNI calls, including varargs, in both directions (Java -> C, C -> Java)
- Reflection (method invocation)
- Exception delivery across native and JVM stack frames
- Multithreading support
- Locking/Mutex support
- Handling unresolved classes (i.e., dynamic bridge support)
- …
Non-optimizing JIT compiler is mostly feature-complete:

- Currently only targets RV64G
- Passes 65 of 68 core tests

Next steps:

- Fix the last bugs/corner cases
- Run JikesRVM on actual hardware (RocketChip with BOOM)
- Port the optimizing compiler
- Integrate updates to support JDK 8 and OpenJDK class library
- Upstream the port (2017)
• JikesRVM enables new managed-language research
• We are using it to investigate hardware support for pause-free Garbage Collection
JikesRVM Port Runs Actual Programs...

...but still has a few bugs
Please get in touch if you would like to get involved!

maas@eecs.berkeley.edu